



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

(Laboratorio di)
Amministrazione di sistemi

Shell: gestione dei processi

Marco Prandini

Dipartimento di Informatica – Scienza e Ingegneria

Convenzioni

- Il font `courier` è usato per mostrare ciò che accade sul sistema; i colori rappresentano diversi elementi:
 - `rosso per comandi da impartire o nomi di file`
 - `blu per l'output dei comandi`
 - `verde per l'input (incluse righe nei file di configurazione)`
- Altri colori possono essere usati in modo meno formale per evidenziare parti da distinguere nei comandi o indicazioni importanti nel testo
- I parametri formali sono normalmente scritti in maiuscolo e riportati nello stesso colore nel testo che ne descrive l'utilizzo

Principi di shell scripting

- Bash può essere usata per programmare task da eseguire automaticamente anziché dover impartire comandi a mano
- Ci sono due aspetti importanti da tenere a mente rispetto a un linguaggio di programmazione come C o Java

1) Gli elementi di base gestiti da bash sono file e processi

bash ha come scopo fondamentale l'avvio di processi, la predisposizione delle comunicazioni tra loro e col filesystem, il controllo dello stato in uscita. È fondamentale pensare sempre, quando si scrive o si analizza una riga di comando, a quali processi verranno eseguiti e a quali file possono essere coinvolti

2) Il linguaggio di bash è interpretato, non compilato

Il significato dato a molti caratteri è sintattico, non letterale, e la riga di comando effettivamente eseguita risulta da un procedimento, detto **espansione**, che individua sottostringhe speciali contrassegnate da caratteri speciali, e le sostituisce col risultato di una corrispondente elaborazione

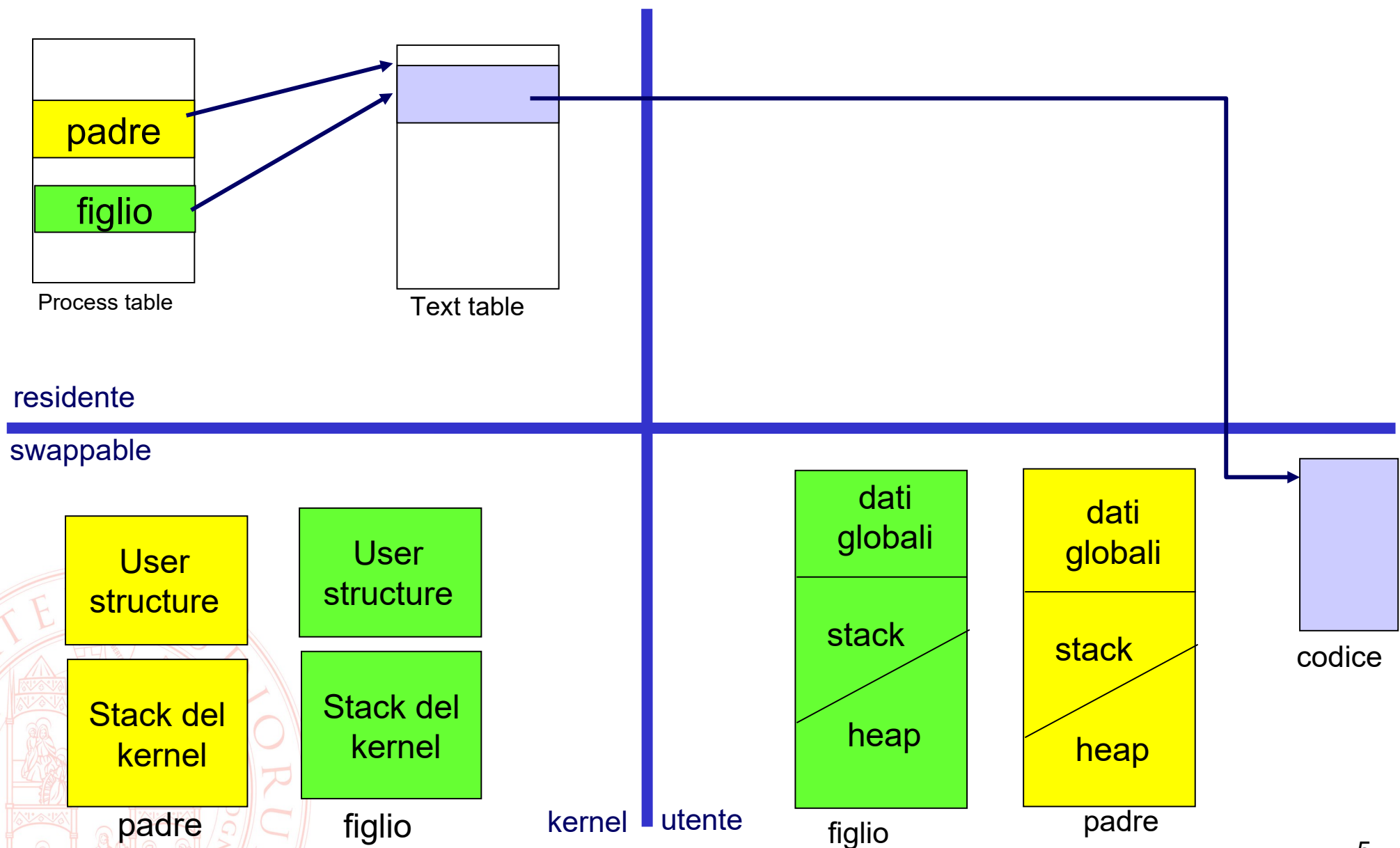
Partiamo dal semplice lancio di programmi

- Ricordiamo come si crea un nuovo processo:
- **fork**: crea una copia “pesante” del processo corrente
 - duplica tutte le risorse
 - condivide il codice
- **exec**: sostituisce il codice del processo padre con quello caricato da un programma
- Quando si lancia un programma quindi la prima cosa che accade è che viene duplicato il processo bash con tutte le sue risorse



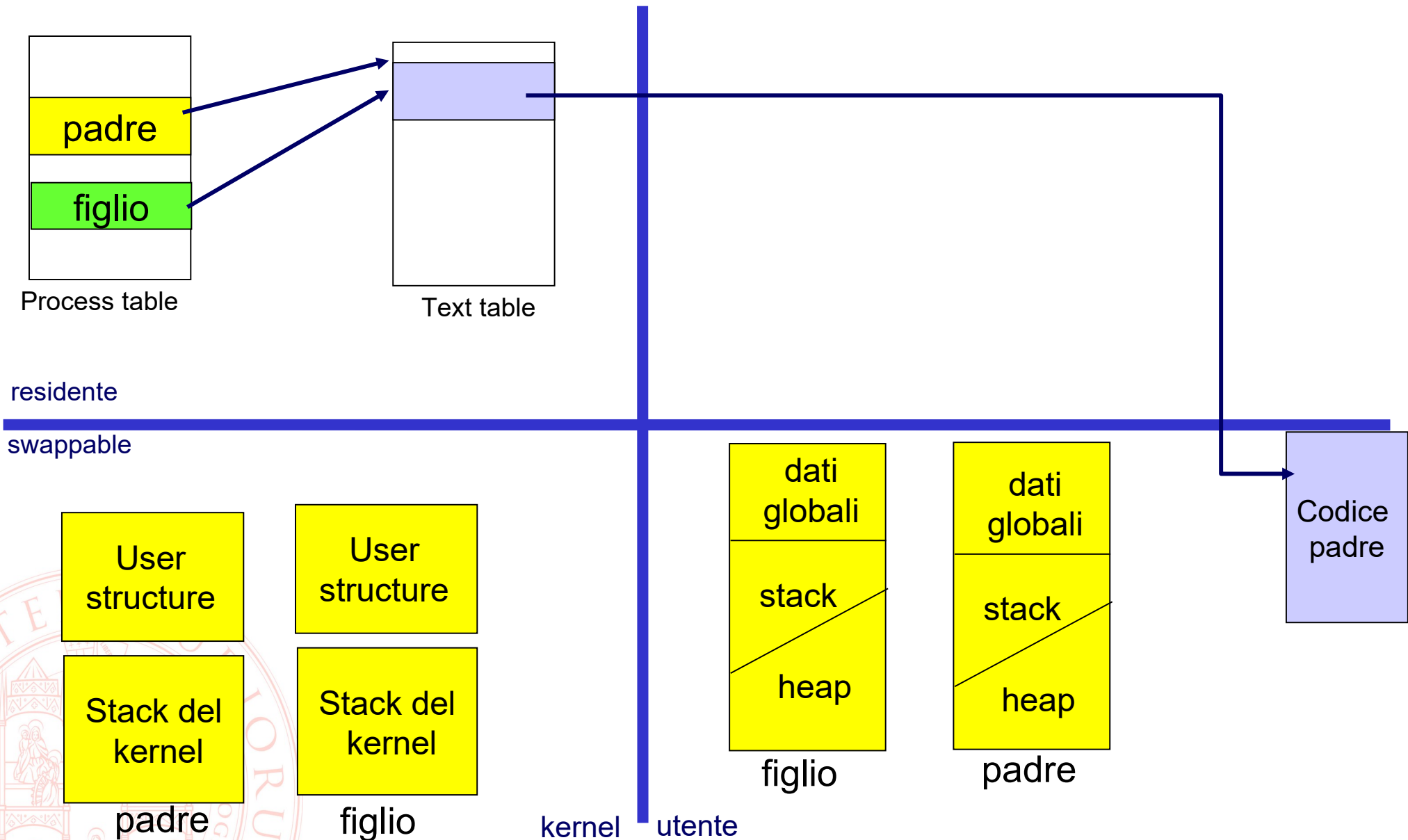
Effetti della fork()

(courtesy: Sistemi Operativi)



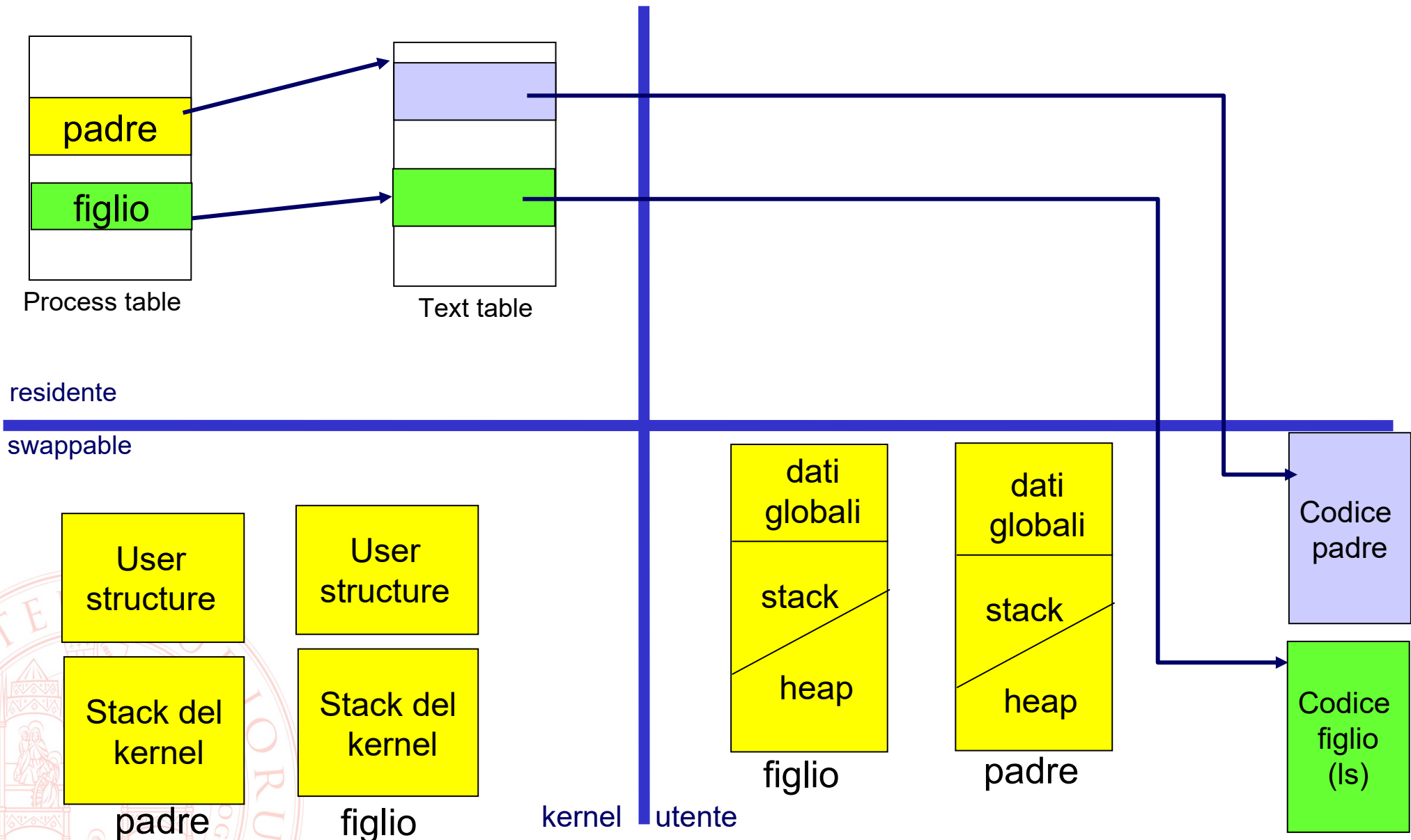
Esempio: effetti della exec() sull'immagine

(courtesy: Sistemi Operativi)



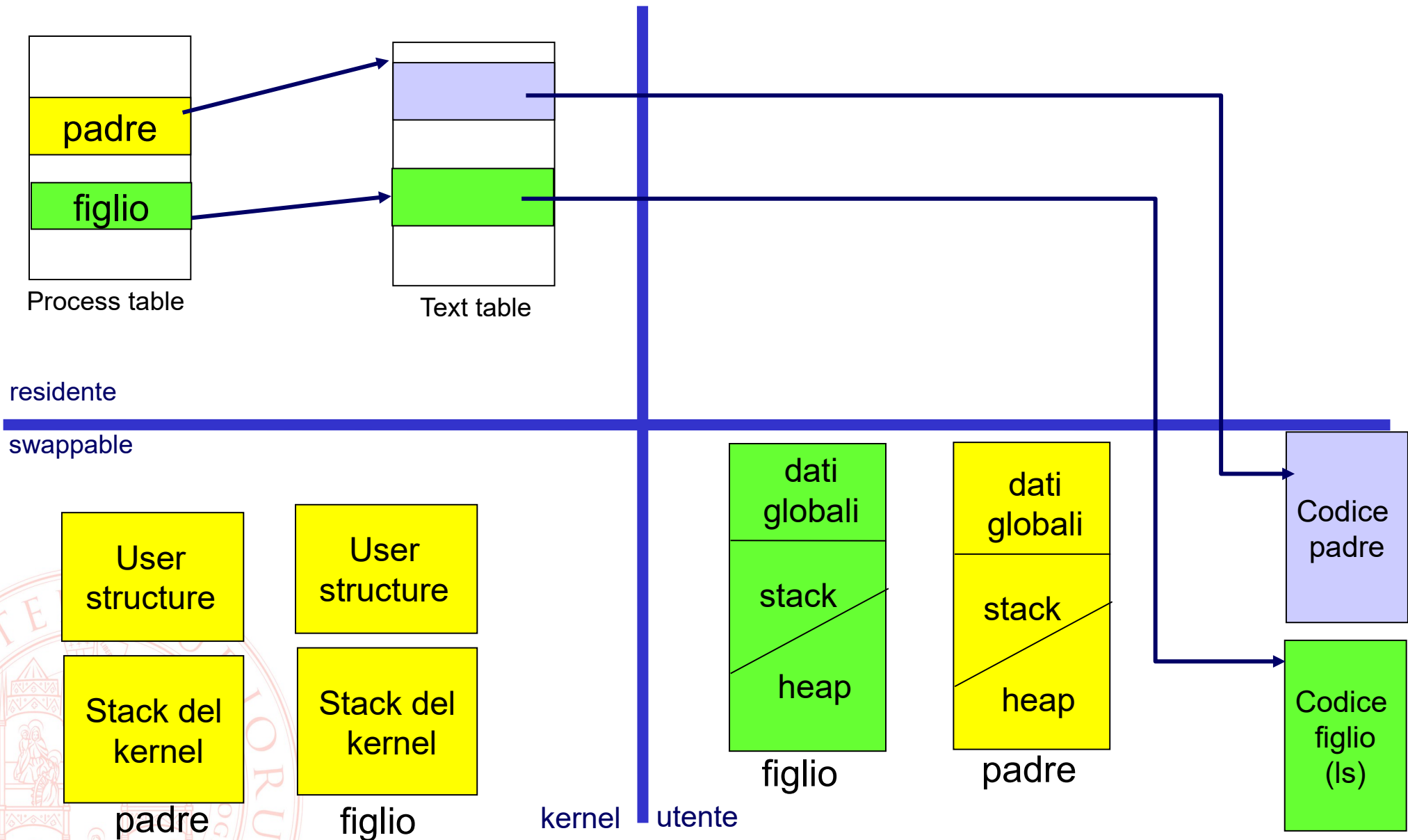
Esempio: effetti della `exec()` sull'immagine

(courtesy: Sistemi Operativi)



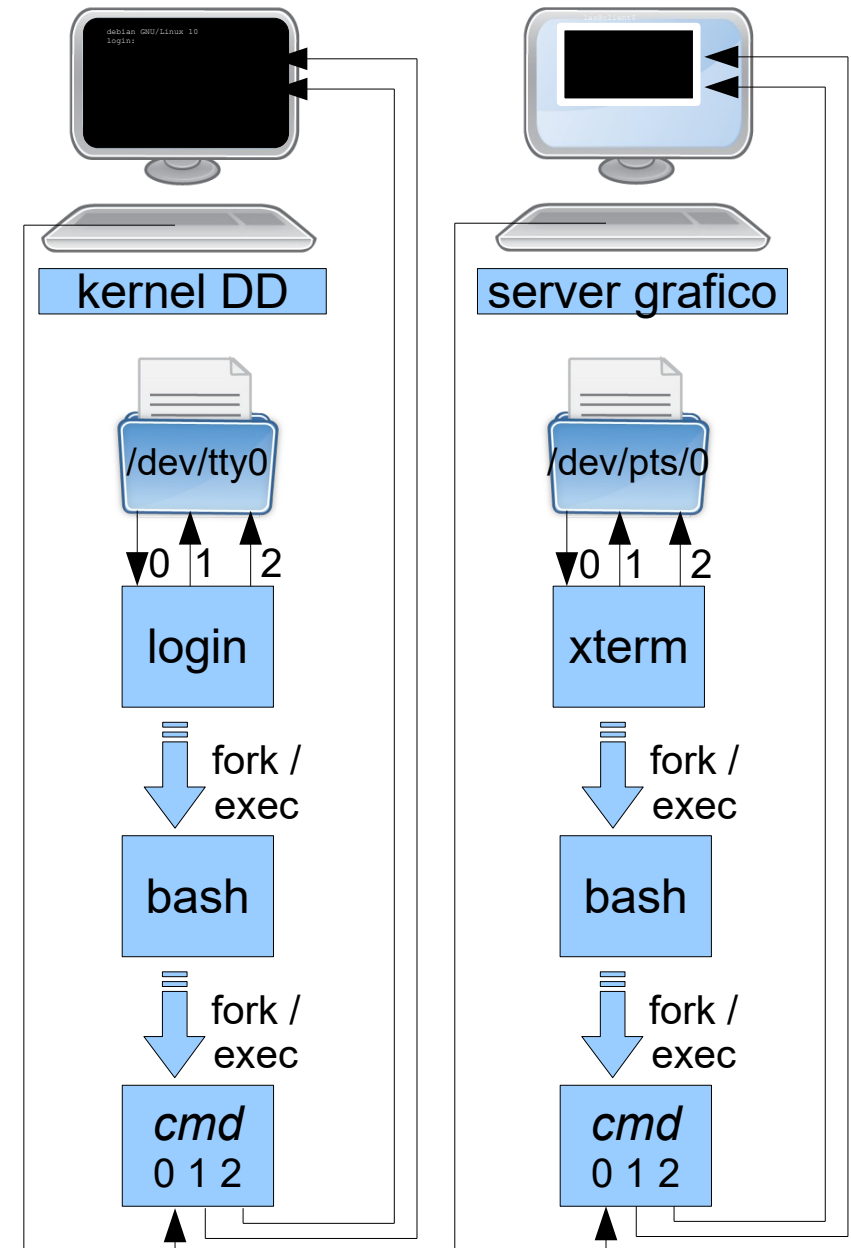
Esempio: effetti della `exec()` sull'immagine

(courtesy: Sistemi Operativi)



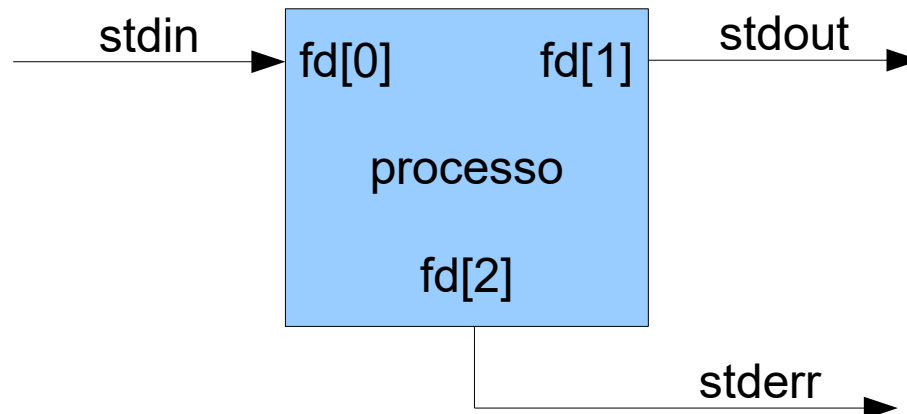
Stream, shell, terminale e lancio di programmi

- All'avvio il kernel inizializza i dispositivi HW e li espone come
 - /dev/tty* (terminali virtuali che accedono direttamente a console)
 - /dev/pts/* (terminali che accedono a finestre grafiche)
- Il device driver che gestisce tali file
 - vi rende disponibili per la lettura i caratteri digitati da tastiera
 - preleva i caratteri che vi vengono scritti e li visualizza a schermo
- Viene avviato un processo di gestione del terminale
 - apre in lettura il file speciale
→ assegnato file descriptor 0
 - apre in scrittura due volte il file speciale
→ assegnati file descriptor 1 e 2
- “qualcuno” istruisce l'avvio di bash
 - eredita i f.d. quindi comunica col terminale
- si lancia un comando da bash
 - (di default) eredita i f.d. quindi comunica col terminale



Elementi di base – stream e ridirezione

- Per convenzione quindi tutti i comandi *nix che operano su stream di testo (filtri) sono progettati per disporre di tre stream con cui comunicare con il resto del sistema:
 - standard input in ingresso (file descriptor 0)
 - standard output in uscita (file descriptor 1)
 - standard error in uscita (file descriptor 2)



Premessa: shell expansion

La shell opera secondo un procedimento di *espansione*

- Individua sequenze speciali contrassegnate da *meta-caratteri*, che non vengono presi a valore nominale
- Interpreta il significato della sequenza speciale
- Al posto della sequenza mette il risultato dell'interpretazione, creando una riga di comando diversa da quella digitata
 - Se un'espansione fallisce (ad esempio la sequenza speciale è mal formata, o dipende dalla presenza di dati che a tempo di esecuzione mancano) la sequenza è solitamente lasciata inalterata sulla riga di comando
- Ci sono ben 12 passi che svolgono manipolazioni diverse della riga di comando, in una sequenza precisa
- Alcuni/tutti possono essere saltati per mezzo del *quoting*, cioè proteggendo i meta-caratteri da non interpretare, per mezzo di altri caratteri speciali: apici ' , doppi apici " , backslash \
 - **Uso minimale del quoting: evitare che gli spazi vengano interpretati dalla shell come separatori tra comandi e argomenti**

Riga di comando da espandere

- Ogni comando può essere preceduto da assegnamento di valore a variabili
 - es. **A=40** **mycommand** | **othercommand** > **outfile**
 - **queste parti** vengono temporaneamente accantonate
- Bash passa all'espansione degli **elementi** della riga di comando come descritto nel seguito
- Bash predispone le **ridirezioni**
- Bash riprende gli **assegnamenti** accantonati
 - ogni parte di testo dopo '=' viene sottoposta (vedi seguito) a
 - tilde expansion
 - parameter expansion
 - command substitution
 - arithmetic expansion
 - quote removal
 - e assegnata alla variabile corrispondente
- Vengono eseguiti i comandi

Ridirezione da/verso file

- Bash, nell'interpretare la riga di comando, può **disconnettere gli stream predefiniti dal terminale** (chiudendoli nel processo figlio dopo la fork) e **far trovare gli stessi file descriptor aperti su di un file diverso** (aprendolo prima della exec)
- Ridirezione dello stdout: **>** e **>>**
 - `ls > miofile` scrive lo stdout di ls nel file miofile (troncandolo)
 - `ls >> miofile` scrive lo stdout di ls nel file miofile (in append)
 - se miofile non esiste viene creato
- Ridirezione dello stderr: **2>** e **2>>**
 - come sopra ma ridirige lo stderr
- Confluenza degli stream
 - `ls > miofile 2>&1` ridirige lo stderr dentro stdout e poi stdout su file
l'ordine è importante!
- Ridirezione dello stdin **<**
 - `sort < miofile` riversa il contenuto di miofile su stdin di sort

Ridirezioni speciali

- Here documents – inviare direttamente un testo a un comando

comando <<MARCATORE

questo testo
va tutto
a finire
sullo stdin
di comando

MARCATORE

- Per una singola linea non serve il marcatore

comando <<< "testo da mandare a stdin di comando"



Ridirezioni speciali

- Se si vogliono ridirigere stream definitivamente si può usare **exec [ridirezione]**

Es. **exec 2>/dev/null**

- tutti i comandi eseguiti da qui in poi avranno stderr riversato su /dev/null
- non pratico interattivamente (la shell usa stderr per mostrare prompt e echo di quel che scrivete!) ma utilissimo negli script
- può essere ripristinato al settaggio originale con **exec 2>&-**

- Con **exec** si possono creare anche nuovi fd

- utile perché i fd aperti vengono ereditati dai processi figli

Es. **exec 3< filein 4> fileout 5<> filerw**

- da ora in avanti
 - ogni lettura dal **fd 3** fatta con **<&3** leggerà da **filein**
 - ogni scrittura fatta con **>&4** sul **fd 4** scriverà su **fileout**
 - il **fd 5** può essere usato sia per leggere che per scrivere su **filerw**
- per chiudere: **exec 3>&- 4>&- 5>&-**

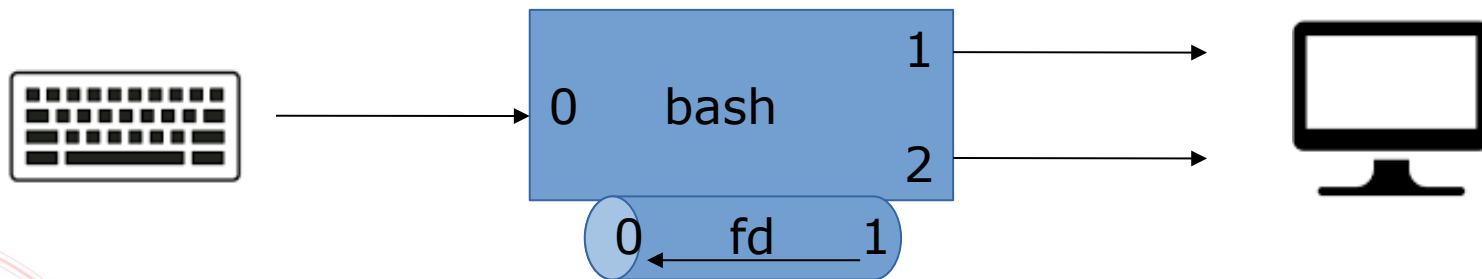
bash pipe

- Cosa succede quando si esegue

`ls | sort`

- Bash prepara il terreno perché ciò che `ls` produce su `stdout` venga riportato su `stdin` di `sort`

1) Call di `pipe (fd[])`



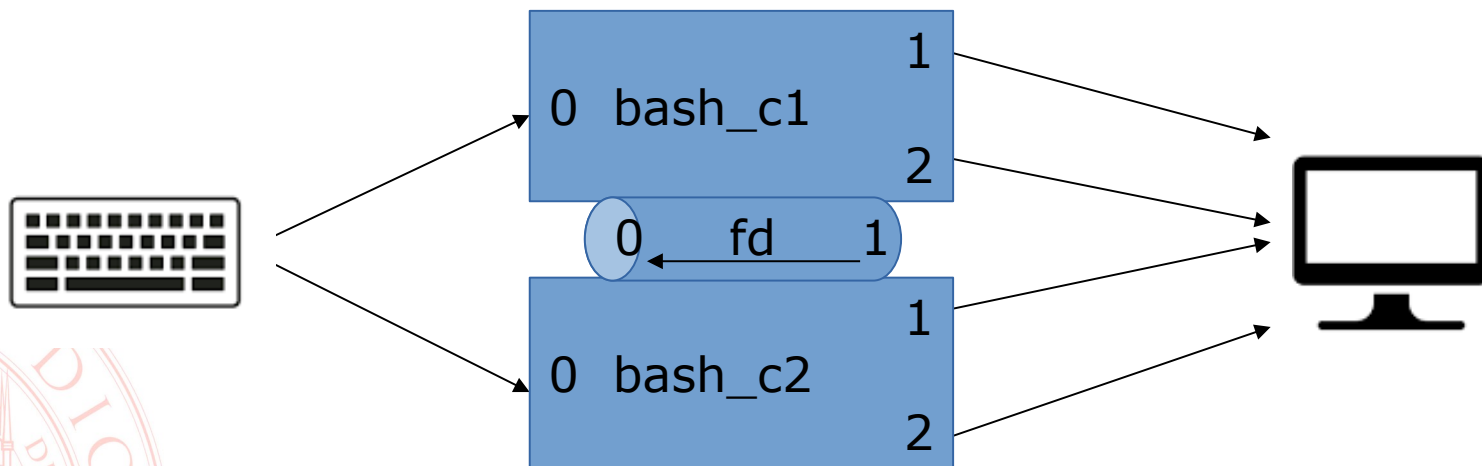
bash pipe

- Cosa succede quando si esegue

`ls | sort`

- Bash prepara il terreno perché ciò che `ls` produce su `stdout` venga riportato su `stdin` di `sort`

2) Call (due volte) di `fork`



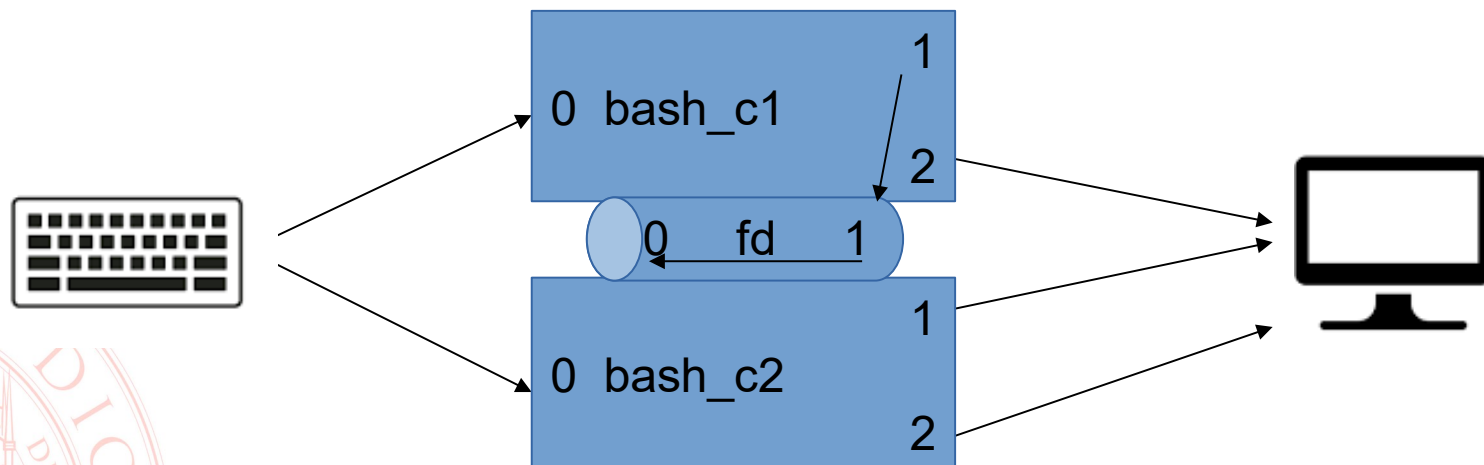
bash pipe

- Cosa succede quando si esegue

`ls | sort`

- Bash prepara il terreno perché ciò che `ls` produce su `stdout` venga riportato su `stdin` di `sort`

3) Child 1 chiama `dup2 (fd[1] , 1)` – taglia lo stream `stdout`, crea un duplicato dell'estremità scrivibile della pipe, e gli assegna il file descriptor 1 (`stdout`)



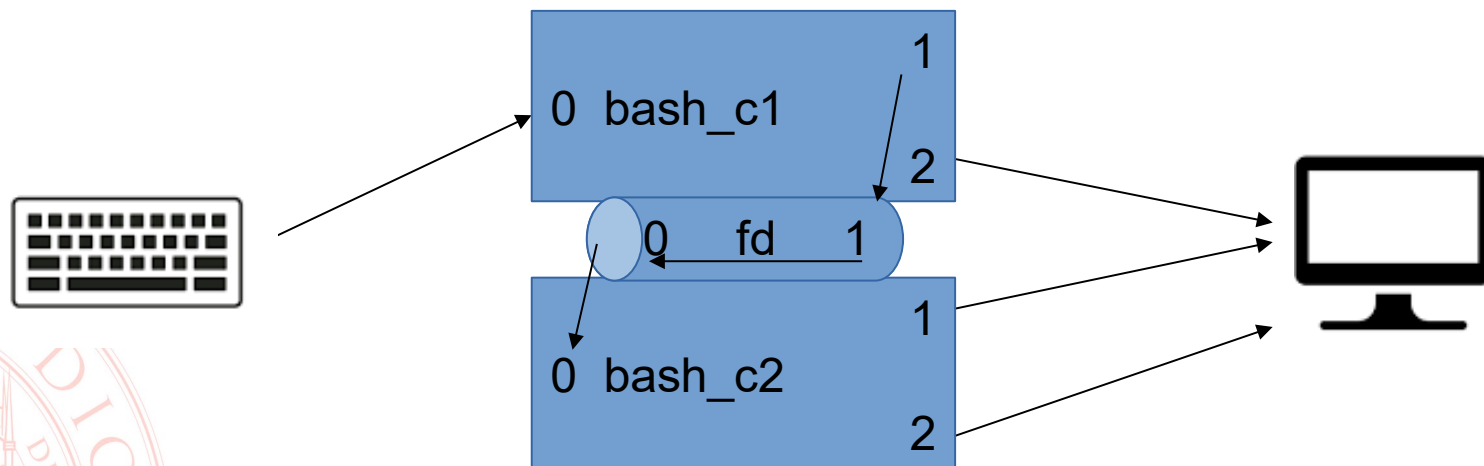
bash pipe

- Cosa succede quando si esegue

`ls | sort`

- Bash prepara il terreno perché ciò che `ls` produce su `stdout` venga riportato su `stdin` di `sort`

4) Child 2 chiama `dup2 (fd[0] , 0)` – taglia lo stream `stdin`, crea un duplicato dell'estremità leggibile della pipe e gli assegna il file descriptor 0 (`stdin`)



5) Child 1 chiama `close (fd[0])` e child 2 chiama `close (fd[1])` per evitare utilizzi incoerenti della pipe

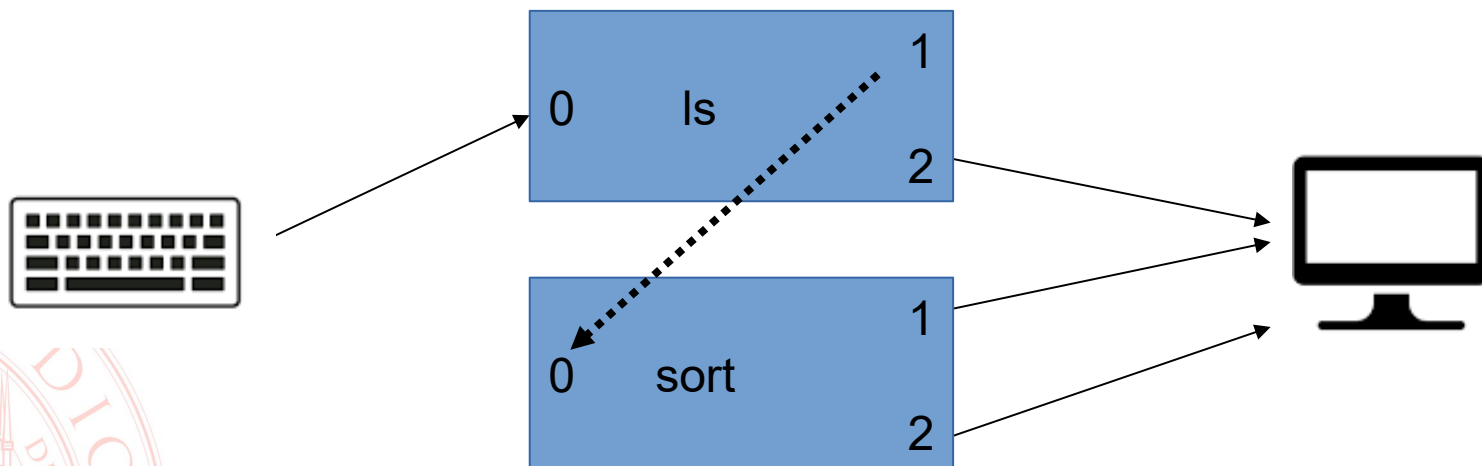
bash pipe

- Cosa succede quando si esegue

`ls | sort`

- Bash prepara il terreno perché ciò che `ls` produce su `stdout` venga riportato su `stdin` di `sort`

6) Child 1 chiama `exec("ls")` e child 2 chiama `exec("sort")` - I nuovi programmi prendono vita e usano i loro stream standard senza bisogno di sapere a cosa sono connessi. Il sistema operativo implementa buffering, sincronizzazione, e segnali per le eccezioni



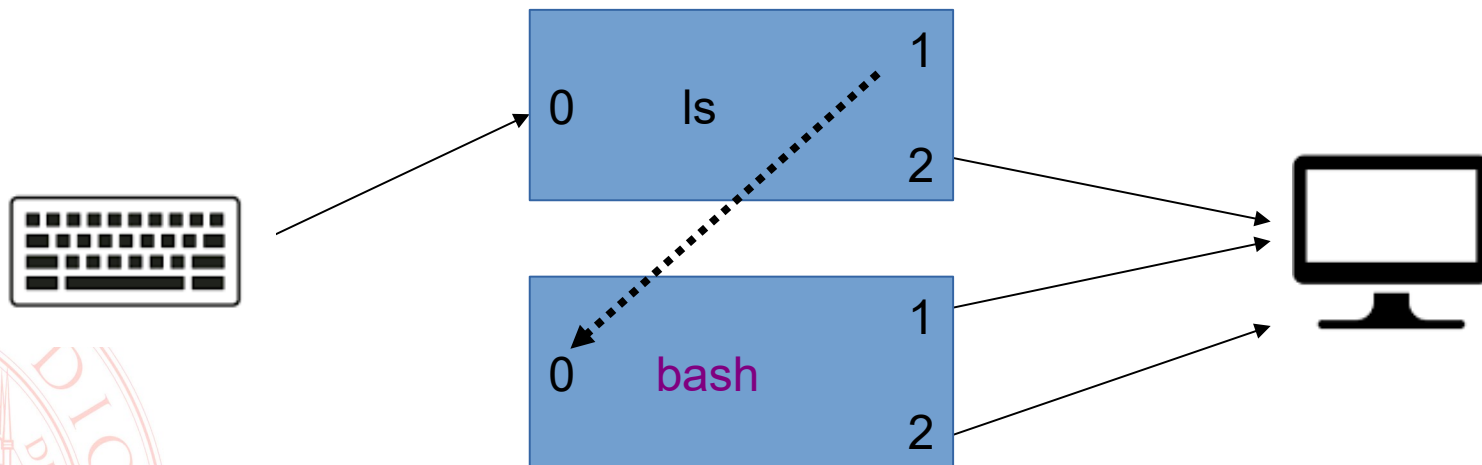
bash pipe + builtin

- Cosa succede quando si esegue

```
ls | builtin_o_funzione
```

- Bash prepara il terreno perché ciò che ls produce su stdout venga riportato su stdin di ... **bash!**

6-variant) Child 1 chiama `exec("ls")` e child 2 non esegue `exec`, perché builtin e funzioni devono essere interpretate da una shell. Questa shell “figlia” è comunque un processo separato a tutti gli effetti (proprio spazio di memoria, propri stream di input, output, error)



Subshell

- Una pipeline che contiene un builtin crea una *subshell* implicita
- È possibile forzare la creazione di subshell per far eseguire sequenze di comandi nello stesso processo bash

```
( comando1 ; comando2 ; ... )
```

- nota: il ; è equivalente a un “a capo”

- Il processo creato

- ha il proprio spazio di memoria
- tutto ciò che viene dato sullo stdin della subshell è disponibile sullo stdin dei comandi
- tutto ciò che i comandi producono su stdout e stderr è prodotto dagli stream corrispondenti della subshell

```
producer | ( step1 ; step2 ; step3 ) 2>/dev/null | consumer
```



Interazione coi processi

- Ogni comando lanciato da shell o dal sistema diviene un **processo**. I processi sono identificati
 - globalmente nel sistema da un numero univoco (Process ID o **PID**)
 - in aggiunta, in alcuni casi, da un **Job ID** valido localmente alla shell
- Un processo **svolge le proprie azioni a nome dell'utente che lo ha lanciato** (i processi lanciati da root hanno il potere di assumere l'identità di altri utenti, così facendo si “declassano” e perdono il potere di tornare indietro)
- I processi anche non lanciati da una stessa pipeline possono comunicare tra loro
 - per mezzo di sistemi da predisporre appositamente (named pipe, socket)
 - in modo più limitato ma semplice per mezzo dei **segnali**



Segnali – caratteristiche di base

■ I segnali sono **eventi asincroni** notificati dal kernel a un processo

- generati dal kernel stesso (eventi hardware, eccezioni durante IPC, ecc.)
- generati da un altro processo
 - eseguito dallo stesso utente del destinatario (o da root)
- Il contenuto informativo è limitato a un numero

■ Ricezione:

- il controllo dei segnali ricevuti avviene ogni volta che il processo rientra in user space (es. dopo una syscall o quando schedato sulla CPU)
- se tra un controllo e il successivo sono stati ricevuti più segnali diversi, vengono posti in uno stato “pending”
 - l'ordine in cui verranno presi in considerazione non è specificato
 - pending non è una coda: che ne arrivi uno o più (dello stesso tipo) il flag sarà semplicemente settato

■ Gestione (a livello di sistema operativo):

- Ogni processo può “registrare” presso il sistema operativo una **routine di gestione (handler)** per un segnale.
- alla rilevazione di un segnale pending, il flusso di esecuzione del processo a cui è destinato viene interrotto e viene eseguito l'handler
- durante l'esecuzione dell'handler, i segnali dello stesso tipo sono bloccati
 - non causano esecuzioni annidate dell'handler ma settano il flag pending

Signal disposition

- Il comportamento di un processo alla ricezione di un segnale può essere
 - terminare (eventualmente con core dump)
 - ignorarlo
 - sospendersi (stato stop)
 - riprendersi da stop (cont)
- Vedere **signal (7)** per l'elenco delle disposition predefinite
- La disposition può essere modificata da un processo
 - tre possibili scelte
 - attuare quella di default
 - ignorare il segnale
 - eseguire un handler
 - fanno eccezione i segnali KILL e STOP, che non possono essere bloccati, ignorati, o intercettati da un handler personalizzato



Handler in bash

- Il builtin **trap** permette di definire un'azione personalizzata da eseguire alla ricezione di un segnale.

```
trap [-lp] [[codice_da_eseguire] segnale ...]
```

- Oltre ai segnali standard, bash riconosce pseudo-segnali per il debugging degli script:
 - **DEBUG** è lanciato dalla shell prima di eseguire ogni comando
 - **RETURN** è lanciato dalla shell
 - dopo il rientro da una chiamata a funzione
 - dopo l'inclusione di un file con **source**
 - **ERR** è lanciato dalla shell ad ogni comando che fallisce
 - **EXIT** è lanciato dalla shell in uscita (sia causata da **exit**, **fine script**, o qualsiasi segnale di terminazione - tranne ovviamente **KILL**)

- **NOTA1:** i signal handler non vengono ereditati dai processi figli
- **NOTA2:** l'esecuzione di un handler non blocca i segnali dello stesso tipo
- **NOTA3:** quando bash esegue un comando, il processo bash non è schedulato fino alla terminazione del child → non vengono controllati i segnali

Invio di segnali

- Per inviare un segnale a un processo si può usare

`kill [options] <pid> [...]`

- PID negativi identificano l'intero process group
- l'opzione `-l` / `-L` elenca i segnali supportati

- Il terminale trasforma la ricezione di alcune combinazioni di tasti in segnali inviati al processo che lo sta occupando:

`Ctrl + Z` → `SIGTSTP`

`Ctrl + C` → `SIGINT`

`Ctrl + \` → `SIGQUIT`

- osservazione a lato: il terminale genera anche altri effetti di controllo non legati ai segnali, come `eof = ^D`; `start = ^Q`; `stop = ^S`;

sleep

- Il comando **sleep** innesca un timer per far “dormire” il processo
- Il parametro può essere un float
 - di default interpretato in secondi
 - sono supportati i suffissi **m**(inutes) **h**(ours) **d**(ays)
- Interazioni coi segnali – valgono le regole di qualsiasi altro comando lanciato dalla shell
 - sleep è un comando esterno
 - genera un processo figlio
 - **mandare un segnale alla shell che lo ha lanciato non lo tocca**
 - sleep invoca una system call che sospende il processo
 - fino al termine della sleep il processo non rientra in user mode
 - **i segnali sono ricevuti ma non processati**



Processi in background

- Si può usare un'unica shell per l'esecuzione contemporanea di più comandi che non abbiano necessità di accedere al terminale, lanciandoli in **background** (sullo sfondo).
- Questo si ottiene postponendo il carattere **&** alla command line.
 - La shell risponde comunicando un numero tra parentesi quadre (**job id**) che identifica il job **localmente** a questa shell.
 - per usarlo al posto di un PID, si utilizza **%job_id**
 - **MOLTO UTILE:** Il PID del processo viene memorizzato nella variabile **\$!**
- Se si lancia una command line senza **&**, e si vuole rimediare, si può dare un segnale di **STOP** con Ctrl+Z.
 - Anche in questo caso si riceve un job id.
 - Con il comando **bg %job_id**, si invia un segnale CONT che riavvia il processo e contemporaneamente lo si mette in background.

wait

- Il builtin **wait** permette di bloccare l'esecuzione fino al completamento dei job in background
 - di default attende il completamento di tutti i job
 - si possono passare come argomento **job_id** specifici
- Se durante l'attesa la shell riceve un segnale per il quale è definito un handler con **trap**
 - **wait** esce immediatamente con exit code > 128
 - l'handler viene eseguito
 - l'esecuzione prosegue dopo la wait
 - si può controllare in **\$?** l'exit code di wait per capire cosa l'ha terminata



jobs e foreground

- Un processo in background non riceve più comandi dal terminale, poiché la tastiera torna ad agire sulla shell;
 - continua però a utilizzare il terminale per STDOUT e STDERR
- se è necessario riportare in **foreground** (primo piano) un processo ricollegandolo così al terminale, si usa il comando **fg %job_id**.
- Il comando **jobs** mostra l'elenco dei job, cioè di tutti i processi avviati dalla shell corrente, indicando il loro stato (attivo o stoppato).
- Per esempi e approfondimenti sulla propagazione di segnali a child process:

<https://linuxconfig.org/how-to-propagate-a-signal-to-child-processes-from-a-bash-script>

Modificatori per processi in bg

- **nohup** <comando> evita che la shell, alla chiusura, invii il segnale SIGHUP al <comando> (il che normalmente ne causerebbe la terminazione)
 - provvede, inoltre, a scollegare l'output del processo dal terminale se non fatto esplicitamente nell'invocazione.
 - di default, nohup dirige l'output sul file 'nohup.out'
- **nice** <comando> lancia <comando> con una *nice*ness diversa da zero, modificando la priorità del processo
 - di default 10
 - valori negativi (che incrementano la priorità) sono utilizzabili solo da root
- **disown** rimuove completamente un job dalla job table della shell
 - di default quello lanciato per ultimo
 - con l'opzione **-h** implementa anche l'immunità all'hangup
- **Note:**
 - **nice** e **nohup** sono comandi esterni e usati all'avvio di un processo, anche insieme
es. **nice nohup long_calculation &**
 - **disown** è un builtin che agisce su PID/job_id di processi lanciati in precedenza